# A Fast Heuristic Algorithm for Redundancy Removal

Maxim Teslenko
Ericsson Research
Ericsson AB
164 80 Stockholm, Sweden
maxim.teslenko@ericsson.com

Elena Dubrova
Dept. of Electronic and Embedded Systems
Royal Institute of Technology - KTH
164 40 Stockholm, Sweden
dubrova@kth.se



***Abstract*—Redundancy identification is an important step of the design flow that typically follows logic synthesis and optimization. In addition to reducing circuit area, power consumption, and delay, redundancy removal also improves testability. All commercially available synthesis tools include a redundancy removal engine which is often run multiple times on the same netlist during optimization. This paper presents a fast heuristic algorithm for redundancy removal in combinational circuits. Our idea is to provide a quick partial solution which can be used for the intermediate redundancy removal runs instead of exact ATPG or SAT-based approaches. The presented approach has a higher implication power than the traditional heuristic algorithms, such as FIRE, e.g. on average it removes 37% more redundancies than FIRE with no penalty in runtime.**

## I. Introduction

Combinational redundancy removal is an optimization problem that can be formulated as follows: A gate or a net in a combinational circuit is *redundant* if it can be removed without changing the functionality of the circuit. Very few, if any, synthesis tools guarantee that the circuits they produce do not contain redundancy. Unnecessary gates or connections are usually introduced by the traditional optimization techniques such as factorization [1] or local transformations [2]. In principle, it is possible to restrict the transformations applied by a synthesis tool to those that preserve non-redundancy of the original circuit. However, it has been shown that redundancy gives an algorithm a greater flexibility in restructuring the logic and more possibilities to find a better implementation [3].

Since redundancy cannot always be avoided, all commercially available synthesis tools include a redundancy removal engine that may be used multiple times on the same netlist during optimization. The presence of redundancy can cause several problems. First, redundancy increases chip area, and may increase its power consumption and propagation delay [4]. Second, redundancy is the reason for undetectable faults in combinational circuits. Although undetectable faults do not affect the operation of the circuit, they may block the detection of other faults and may invalidate the completeness of a test set that was generated [5].

In this paper, we consider two types of redundancy: (1) redundancy associated with undetectable stuck-at faults, which do not cause incorrect output values for any input assignment, and (2) functional duplication, which occurs if different gates implement the same function. *Automatic test patter generation (ATPG)* and *fault-independent* methods target the first type of redundancy.

ATPG-based algorithms use exhaustive test pattern generation to prove the undetectability of faults on redundant lines [6], [7], [8], [9]. They guarantee detection of all such faults, but they have the exponential worst-case time complexity.

Fault-independent methods analyze the topology of a circuit without targeting a specific fault. This can be done either by an explicit analysis of reconvergent fanout regions, as in [10] and [11], or by propagating uncontrollability and unobservability values, as in FIRE [12] and its extensions [13], [14], [15]. Although fault-independent methods cannot determine all undetectable faults, they have an advantage of the polynomial worst-case time complexity.

*Satisfiability checking (SAT)* [16], *Binary Decision Diagram (BDD) sweeping* [17] and *structural hashing* [18] methods target the functional duplication type of redundancy. SAT-based algorithms usually first partition all gates into equivalence classes by random simulation, and then apply satisfiability check for each pair in the class to verify equivalence. BDD-sweeping algorithms build a binary decision diagram for every gate in the circuit and merge gates with equivalent BDDs. Both, SAT and BDD-sweeping, guarantee detection of all functional duplications, but they have the exponential worst-case time complexity. Structural hashing can identify structurally isomorphic equivalent vertices in the linear time.

This paper presents a redundancy identification and removal algorithm which employs fault-independent search strategy introduced in the redundancy identification algorithm FIRE [12]. FIRE identifies undetectable faults which require conflicting value assignments on a single line in the circuit for their detection.

Some other extensions of FIRE have been proposed. In [13], conflicting value assignments for pairs of vertices rather than single vertices are considered. In [14], a technique for maximizing conflicting value assignments on a single vertex is presented. A large number of direct and indirect logic implications are derived and stored in an implication graph. These implications are used to increase the implication power of FIRE. In [15], binary resolution in addition to static logic

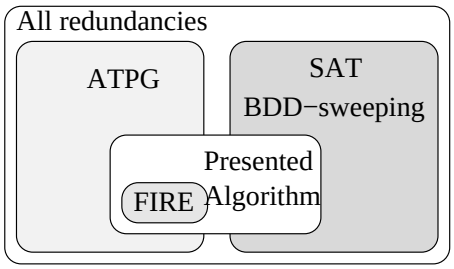


Fig. 1: Redundancy identification coverage of different methods.

implications are used for maximizing conflicting value assignments on multiple vertices. All approaches described above allow for identification of more undetectable faults compared to FIRE, but they make the complexity prohibitive for large circuits.

The presented algorithm differs from FIRE in several aspects. The first improvement is an increased implication power. A fundamental difference of the presented approach from other extensions of FIRE [13], [14], [15] is that we do not perform any extra search to find additional implications. Rather, we re-use the information which is anyway available in the algorithm's flow.

The second improvement is the ability to identify some vertices which implement equivalent or complemented functions. Similarly to the previous improvement, this is done with a minimum search, by re-using the information from the algorithm's flow. This improvement allows us to find some redundancies which cannot be found by an ATPG-based approach (see Figure 1). To our best knowledge, the presented technique is the first which can find structurally different equivalent vertices in a circuit in less than exponential time.

The runtime improvements include a reduced number of unobservability checks during unobservability propagation stage and a special treatment of vertices with a single input. Overall, the proposed improvements allow us to find 37% more redundancies than FIRE without increasing its runtime.

Another difference from FIRE is that the presented algorithm removes redundancy, while FIRE is only an identification algorithm. The fundamental problem of redundancy removal is to keep implication database updated after a net or gate has been removed from the circuit. In the worst case, the complete database has to be re-calculated. We use properties of indirect implications which allow us to update the implications database instantly.

The paper is organized as follows. Section II introduces the notation and definitions used in the sequel. Section III gives the background on FIRE algorithm. Section IV presents the new algorithm. Section V describes in details new contributions and differences between our approach and the one of FIRE. Section VI summarizes experimental results. Section VII concludes the paper and discusses open problems.

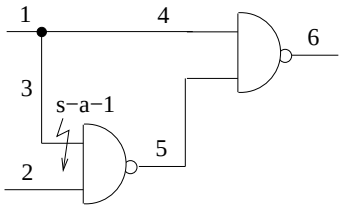


Fig. 2: An example of undetectable fault.

## II. Notation and Definitions

*Undetectable fault* is a fault which cannot be detected by any input pattern [6]. This can happen either because it is not possible to apply to the faulty line the value opposite to the value of the fault (*uncontrollability*), or because the effect of the fault cannot be propagated to the output (*unobservability*). In a combinational circuit, undetectable stuck-at faults are always caused by redundancy [12].

Let $C = (V,E)$ be a Boolean circuit, where $V$ represents gates and primary inputs and $E$ describes the nets connecting the gates. We use the letters $v,u$ and $w$ to denote the vertices of $C$. The letters $s$ and $q$ are designated for the *stem* of a multiple fanout net, and $b_1,\ldots,b_r$ for its *branches* (as in Figure 4).

The set of predecessors of a vertex $v \in V$ is denoted by $IN(v) = \{u \in V \mid (u,v) \in E\}$. The set of successors of $v$ is denoted by $OUT(v) = \{u \in V \mid (v,u) \in E\}$.

A value on the input of a gate is *controlling*, if its presence determines the value of the gate's output, independently of the values of other inputs. A value on the output of a gate is *controlled* if it was set by a controlling input value. For AND (OR) the controlling and controlled values are the same, namely 0 (1). For NAND (NOR) they are 0 (1) and 1 (0), respectively. The XOR has no controlling and controlled values.

A logic implication is *direct* if it relates inputs and output of a single gate and it is evident from the type of this gate only. For example, 0(1) at one of the inputs of an AND(OR) directly implies 0(1) on gate's output; 1(0) at the output of an AND(OR) directly implies 1(0) at all inputs of the gate.

## III. FIRE Algorithm

FIRE algorithm [12] classifies a stuck-at fault on the line $l$ as undetectable if this fault requires the presence of both, 0 and 1 values (i.e. a conflict) on some other line $r$ as a necessary condition for its detection. All stems in the circuit are checked as candidates to be such a line $r$. For each stem $q$, two sets of faults, $set_0$ and $set_1$, are computed. The set $set_i$ is defined as the set of faults that require $q$ to have value $i \in \{0,1\}$ as a necessary condition for their detection. The set of undetectable faults is obtained by intersecting $set_0$ and $set_1$.

As an example, consider the circuit shown in Figure 2. Stuck-at-1 fault on line 3 requires the value 1 on line 1 for observability and the value 0 on line 1 for controllability. Therefore, this fault in undetectable.

In order to compute $set_i$ for a stem $q$ the FIRE algorithm does the following. The value $\bar{i}$ is set on $q$ and constant

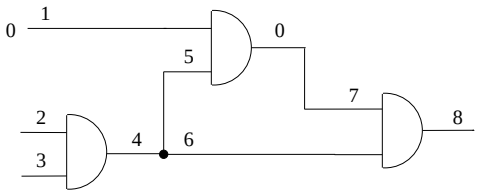


Fig. 3: Example showing that the conditions formulated in Lemma 1 are sufficient but not necessary.

propagation using direct implications is applied recursively. If some line $l$ is assigned the value 1(0), it is uncontrollable for the value 0(1), so stuck-at-1(0) fault at $l$ is added to $set_i$. This is because the line $l$ cannot be assigned value 0(1) and thus cannot be tested for stuck-at-1(0) fault when $q$ has the value $\bar{i}$. Propagation of constants may result in some lines becoming unobservable. If one input of a gate is set to the controlling value, then all other inputs of this gate become unobservable. If some line $l$ is unobservable, then both stuck-at-0 and stuck-at-1 at $l$ are added to $set_i$. The unobservability is propagated backward. If all fanout branches of a stem $s$ are marked unobservable, the following Lemma is applied to decide whether $s$ is also unobservable or not.

*Lemma 1:* [12] A stem $s$ with all its branches marked as unobservable may also be marked as unobservable if, for each branch $b$ of $s$, there exists at least one set of lines $\{l_b\}$ such that the following conditions are satisfied:

1) the branch $b$ is unobservable because of uncontrollability indicators on every line in $\{l_b\}$, and
2) every line in $\{l_b\}$ is unreachable from $s$.

The conditions formulated in Lemma 1 are sufficient, but not necessary, conditions for unobservability. An example is shown in Figure 3. Suppose that the line 1 has the value 0. Then, both fanout branches of the net 4 are unobservable. For the branch 5, the line 1 satisfies both conditions of Lemma 1, since 5 is unobservable because of the 0 value on the line 1, and 1 is unreachable from 4. The branch 6 is unobservable because of the 0 value on the line 7, however, 7 is reachable from 4. So, the second condition of Lemma 1 does not hold and therefore the net 4 is not recognized by FIRE as unobservable.

## IV. Redundancy Removal

The overall flow of presented algorithm is similar to the one of FIRE. The pseudo-code is shown in Figures 5 and 6. In this section, we describe the implementation details necessary for its understanding. In the Section V, we focus on the conceptual differences between the presented algorithm and FIRE.

In the main loop (steps 8 - 58), our algorithm iterates through all vertices of the circuit, $v_{base} \in V$, sets them to the value $i$, $i \in \{0,1\}$, and performs uncontrollability and unobservability propagation in order to compute the information similar to $set_i$ for the stem fed by $v_{base}$ in FIRE algorithm. We use the term *ith run for* $v_{base}$ to refer to the uncontrollability and unobservability propagation with the vertex $v_{base}$ being set to the value $i$, $i \in \{0,1\}$.

In our implementation, every vertex $v \in V$ has the following fields:

1) $unobservable\_outputs(v,i) \in \{0,1,\ldots,|OUT(v)|\}$ is the number of outgoing edges of $v$ which become unobservable after $i$th run.
2) $master(v,i) \subset IN(v) \cup \{NULL, ALL\}$ is a pointer to the predecessor of $v$ which has the controlling value for $v$ in the $i$th run. Also, $master(v,i) = u$ means that all inputs of $v$, except $u$, are unobservable. If there is more than one such predecessor, $master(v,i)$ is set to the dummy vertex $ALL$, which means that all inputs of $v$ are unobservable. Initially, when vertices are not set to values, $master(v,i)$ is set to $NULL$, which means that no input of $v$ is unobservable.
3) $visited(v) \in \{0,1\}$ shows whether $v$ has been visited or not during unobservability propagation and check, $visited(v) = 1$ if visited, 0 otherwise.
4) $indirect\_implications(v,j)$, $j \in \{0,1\}$, contains the list of pairs of type $(u,k)$, $u \in V$, $k \in \{0,1\}$, such that there exists a logic implication $(v = j) \rightarrow (u = k)$.
5) $invalid\_implication(u) \in \{on, off\}$ shows whether all indirect implications which imply $u$ to some value are valid or not. If $invalid\_implication(u) = on$, then all implications of type $(v = j) \rightarrow (u = k)$ stored as $(u,k) \in indirect\_implications(v,j)$ for some $v \in V$, $j,k \in \{0,1\}$, are not valid.

For each $v_{base}$, the presented algorithm first performs constant propagation and indirect implications learning (steps 13-22, explained in subsection 5.3), then eliminates duplicated and constant vertices (steps 23-40, explained in subsection 5.4), does unobservability propagation (steps 41-46) and finally removes the identified redundancies (steps 47-55).

The procedure PROPAGATEUNCONTROLABILITY performs constant propagation, i.e. it recursively applies direct and learned indirect implications following from the assignment $v_{base} = i$, $i \in \{0,1\}$. The obtained values are stored not only with vertices, but also in the queue $Q(i)$ defined by

$$Q(i) := \{(u,j) \mid u = j \text{ after } i\text{th run for } v_{base}\}.$$

If the justification of $v_{base}$ to $i$ causes a contradiction (i.e some vertex needs to be assigned to different values), then PROPAGATEUNCONTROLLABILITY returns Ø. PROPAGATEUNCONTROLABILITY also fills the field $master(v,i)$ for all gates $v \in V$ in correspondence with its definition.

The procedure OVERAPPROXIMATEUNOBSERVABILITY INIT$(v,i)$ initiates the process of unobservability propagation. It is invoked at all vertices which are set to controlled values. The procedure CHECKUNOBSERVABILITY$(v,u,i)$ checks whether a given edge $(v,u)$ is really unobservable or not. Specific features of unobservability propagation and checking are discussed in subsection 5.1.

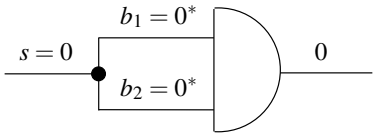


Fig. 4: An example showing that unobservability of all fanout branches does not necessarily imply unobservability of the stem; the sign $''*''$ indicates that the value is unobservable.

## V. Improvements over FIRE

In this section, we describe the improvements of the presented algorithm over FIRE. The first two are runtime improvements, the second two are quality improvements.

### A. Runtime Improvements

*1) Reducing the number of unobservability checks during unobservability propagation:* The first improvement in runtime is achieved by reducing number of unobservability checks during unobservability propagation stage.

The initial source of unobservability in the circuit are vertices which have at least one input set to a controlling value. If an input of a vertex $v$ has a controlling value, other inputs of $v$ become unobservable. The rest of unobservable lines is derived by unobservability propagation.

Usually, if all fanout branches of a stem $s$ are unobservable, $s$ is unobservable as well. However, there are some rare exceptions. For example, consider the circuit shown in Figure 4. Suppose the stem $s$ is set to 0 value. Then, both branches $b_1$ and $b_2$ are unobservable. In spite of this, $s$ is observable.

FIRE resolves the problem with such cases by checking unobservability of a stem $s$ any time all its branches are identified as unobservable (by applying Lemma 1). Since the unobservability check may need to be done multiple times per each run, and number of runs is the number of vertices times two, the number of unobservability checks made by FIRE equals to the number of vertices multiplied by some factor $m$, which represents the average number of unobservability checks per each pair of runs:

$$N^{FIRE}_{unobservability\_checks} = |V| \times m. \quad (1)$$

Even though the factor $m$ can be up to the number of stems in the circuit, our experience is that $m$ is usually small, although larger than 1 in most of the cases.

In the presented algorithm, if we encounter a stem $s$ with all branches being unobservable, we assume unobservability of $s$ without any check, i.e. we overapproximate actual unobservability. The eventual correction is postponed for later. If no redundancy is identified with the overapproximated unobservability, no correction is performed because this unobservability is not going to be used for redundancy removal anyway. This differs our approach from FIRE, where the correction is done always.

The number of unobservability checks needed for the presented algorithm equals to the number of redundancies in the circuit plus the number of incorrectly detected redundancies (caused by the overapproximated unobservability):

$$N_{unobservability\_checks} = N_{redundancies} + N_{incorrect\_redundancies}. \quad (2)$$

In practice, we very rarely have cases where stems are not unobservable if all their branches are unobservable. Thus, we have very few incorrectly identified unobservabilities, and, as a consequence, very few incorrectly detected redundancies. Furthermore, the number of redundancies in a circuit is usually quite small.

We further reduce the number of unobservability checks by using the following property. It shows that a line identified as unobservable by the overapproximation procedure does not need to be checked if it is not set to a constant value.

*Theorem 1:* If during the $i$th run, $i \in \{0,1\}$, a line $l$ is classified as unobservable by the overapproximation procedure and it is not set to a constant value, then $l$ is unobservable.

The Theorem 1 is used at the step 50 of the pseudocode, where the algorithm checks whether the value of $v$ is not set to a constant during the $i$th run. Only if this statement is not satisfied, the unobservability check CHECKUNOBSERVABILITY$(v,u,i)$ is invoked. Note that, in order to reach the step 50, the conditions of **for**-loop at the step 47 need to be satisfied. As a consequence, the unobservability check is done only if the line $(v,u)$ is set to a constant value in both runs. This is possible only if the gate $v$ is either a constant function, or it is equivalent to $v_{base}$ or $\overline{v}_{base}$. Since constant and duplicated functions are removed earlier (at the steps 23-40), unobservability checks have to be performed only if some output of $v_{base}$ or $\overline{v}_{base}$ was identified as redundant. Thus, terms $N_{redundancies}$ and $N_{incorrect\_redundancies}$ in the equation 2 refer not to all identified redundancies, but rather to the redundancies found at the output of $v_{base}$ or $\overline{v}_{base}$. Therefore, the overall number of unobservability checks of the presented algorithm is significantly smaller than the one of FIRE, while the complexity and average runtime of each check is the same for both algorithms.

*2) Special treatment of gates with a single input:* The second improvement in runtime is a result of skipping the outer **for**-loop for the vertices with a single input. All implications which could be obtained during these runs would be equivalent to the ones found during the runs of the algorithm for the only predecessor of the vertex. Therefore, no new redundancies would be identified.

### B. Quality Improvements

*1) Increased implication power:* The first improvement in quality is achieved by re-using the information from the previous runs of the algorithm to increase its implication power in the future runs. This is done as follows.

Suppose that, during $i$th run for the vertex $v_{base}$, the vertices $u_1,\ldots,u_p$ are set to some values $j_1,\ldots,j_p$. We can conclude that the assignment of $v_{base} = i$ implies the values $j_1,\ldots,j_p$ on $u_1,\ldots,u_p$. From this, by using the contrapositive low, we can derive a set of logic implications $(u_1 = \overline{j}_1) \rightarrow (v_{base} = \overline{i}),\ldots,(u_p = \overline{j}_p) \rightarrow (v_{base} = \overline{i})$. Some of them might be indirect

```
algorithm REDUNDANCY REMOVAL(V,E)
   Boolean i, j;
   Vertex v,u,w,v_base;
   Queue Q(0),Q(1);  /* lists of pairs (Vertex v, Boolean i) */
   integer index, p;

1. index := 0;
2. for every vertex v ∈ V in forward topological order do
3.       index(v) := index;  index++;
4.       indirect_implications(v,0) := Ø;
5.       indirect_implications(v,1) := Ø;
6.       invalid_implications(v) := off;
7. end for 2
8. for p from 0 to |V| do
9.       v_base is a vertex with index(v_base) = p;
10.      if |IN(v_base)| = 1 continue  /* Runtime improvement 2 (Section V-A2) */
11.      begin_loop :
12.      For every v ∈ V, set unobservable_outputs(v,i) = 0,master(v,i) = NULL;
/* Uncontrollability propagation */
13.      for every i ∈ {0,1} do
14.            Q(i) = PROPAGATEUNCONTROLABILITY(v_base,i);
15.            if Q(i) is empty then /* could not justify v_base to i */
                     /* stuck-at-ī at the output of v_base is undetectable */
16.                  UPDATEIMPLICATIONS(v_base,v_base);
17.                  replace v_base by the constant ī;
18.                  go to end_loop 57;
19.            for every (v,j̄) ∈ Q(i) do
20.                  add (v_base,ī) to indirect_implications(v,j);
21.            end for 19
22.      end for 13
/* Removal of constant vertices. Quality improvement 2 (Section V-B2) */
23.      for every pair (v,j) ∈ Q(0) ∩ Q(1) do
               /* stuck-at-j at at the output of v is undetectable */
24.            UPDATEIMPLICATIONS(v,v_base);
25.            replace v by the constant j;
26.      end for 23
/* Removal of duplicated vertices. Quality improvement 2 (Section V-B2) */
27.      Create Q̄(1) := {(v,i)|(v,ī) ∈ Q(1)};
         /* Q̄(1) contains all pairs of Q(1) with i being complemented */
28.      for every pair (u,j) ∈ Q̄(1) ∩ Q(0) do
29.            if i = 0 then
30.                  Add u to the equivalence class of v_base, E(v_base);
31.            else
32.                  Add u to the equivalence class of v̄_base, E(v̄_base)
33.      end for 28
34.      Replace all v ∈ E(v_base) by u ∈ E(v_base) closest to primary inputs;
35.      Replace all v ∈ E(v̄_base) by w ∈ E(v̄_base) closest to primary inputs;
36.      if |IN(u)| = 1 and |IN(w)| = 1 then
37.            if u is closer to primary inputs than w then
38.                  Substitute w by an inverter fed by u;
39.            else
40.                  Substitute u by an inverter fed by w;
/* Unobservability propagation */
41.      for every i ∈ {0,1} do
42.      For every v ∈ V, set visited(v) = 0;
43.            for every pair (v,j) ∈ Q(i) do
44.                  if j is the controlled value of v then
45.                        OVERAPPROXIMATEUNOBSERVABILITYINIT(v,i);
46.            end for 43
/* Search for a redundant line */
47.            for every (v,j) ∈ Q(ī) such that unobservable_outputs(v,i) > 0 do
48.                  for every u ∈ OUT(v) do
49.                        if master(u,i) ≠ NULL and master(u,i) ≠ v then
50.                              if the value of v is not set to a constant during ith run or
                                    CHECKUNOBSERVABILITY(v,u,i) = true then
                                    /* stuck-at-j at (v,u) is undetectable */
51.                                 UPDATEIMPLICATIONS(u,v_base);
52.                                 replace (v,u) by the constant j;
53.                                 go to begin_loop 11;
54.                  end for 48
55.            end for 47
56.      end for 41
57.      end_loop :
58.   end for 8
59. end
```

Fig. 5: Pseudo-code of the presented algorithm.

```
algorithm OVERAPPROXIMATEUNOBSERVABILITYINIT(v,i)
   if visited(v) = 0;
         visited(v) := 1;
         for every vertex u ∈ IN(v) − {master(v,i)} do
               CHECKOUTPUTS(u,i)
         end for
end

algorithm CHECKOUTPUTS(v,i)
   unobservable_outputs(v,i)++;
   if unobservable_outputs(v,i) = |OUT(v)| then
   /* Runtime improvement 1 (Section V-A1). Here FIRE does */
   /* unobservability check, while the presented algorithm propagates */
   /* unobservability without any check */
         master(v,i) := ALL;
         OVERAPPROXIMATEUNOBSERVABILITY(v,i);
end

algorithm OVERAPPROXIMATEUNOBSERVABILITY(v,i)
   if visited(v) = 1;
         CHECKOUTPUTS(master(v,i),i)
   else
         visited(v) := 1;
         for every vertex u ∈ IN(v) do
               CHECKOUTPUTS(u,i)
         end for
end

algorithm CHECKUNOBSERVABILITY(v,u,i)
   For every vertex w ∈ V, set visited(w) = 0;
   if ∄w ∈ IN(u) − {v} such that value of w in the ith run is controlling for u
         if unobservable_outputs(u,i) ≠ |OUT(u)| then
               return false;
         else
               if UNOBSERVABILITY(u,i) = false then
                     return false;
   return true;
end

algorithm UNOBSERVABILITY(v,i)
   if visited(v) = 1 then
         return true;
   visited(v) := 1;
   for every u ∈ OUT(v) do
         if ∄w ∈ IN(u) such that visited(w) ≠ 1 and
            value of w in the ith run is controlling value for u then
               if unobservable_outputs(u,i) ≠ |OUT(u)| then
                     return false;
               else
                     if UNOBSERVABILITY(u,i) = false then
                           return false;
   end for
   return true;
end

algorithm UPDATEIMPLICATIONS(v,v_base)
   /* Quality improvement 1 (Section V-B1) */
   UPDATER1(v); /* updating region R_1 */
   for every u such that index(v) < index(u) < index(v_base)
         invalid_implication(u) := on; /* updating region R_2 */
   end for
end

algorithm UPDATER1(v)
   if indirect_implication(v,0) ≠ Ø OR indirect_implication(v,1) ≠ Ø then
         indirect_implication(v,0) := Ø;
         indirect_implication(v,1) := Ø;
         for every u ∈ OUT(v) do
               UPDATER1(u);
         end for
end
```

Fig. 6: Pseudo-codes of the procedures.

implications. For every $r \in \{1,\ldots,p\}$, we store the implication $(u_r = \bar{j}_r) \rightarrow (v_{base} = \bar{i})$ in the field *indirect_implications* of $u_i$ by adding to the field the pair $(v_{base}, \bar{i})$. When we justify the vertex $u_r$ to the value $j_r$ in later runs, these stored

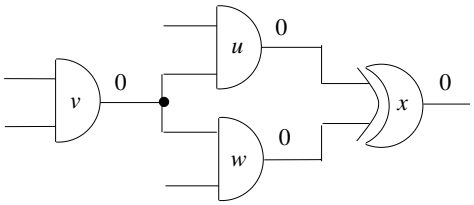


Fig. 7: An example showing how an indirect implication can be obtained.

indirect implications will be used to set the vertex $v$ to the value $k$, for every $(v,k) \in indirect\_implication(u_r, j)$, provided $invalid\_implication(v) = off$.

Note, that indirect implications are not immediately evident from the circuit. They cannot always be derived by FIRE approach. As an example, consider the circuit shown in Figure 7. During 0th run for the vertex $v$, the vertices $u, w$ and $x$ are set to 0 by constant propagation. By using the contrapositive low, we derive logic implications $(x = 1) \rightarrow (v = 1), (u = 1) \rightarrow (v = 1)$ and $(w = 1) \rightarrow (v = 1)$. While the last two are direct, the first one is not. FIRE algorithm would not set the vertex $v$ to 1 when $x$ is justified to 1.

A fundamental difference of the presented algorithm from other approaches which improve FIRE by increasing its implication power [13], [14], [15] is that we do not perform any extra search to find indirect implications. Rather, we re-use information available from the algorithm's flow.

Another important contribution is the following observation. Every time a redundant line is removed, some of the indirect implications become invalid. On one hand, checking whether each indirect implication needs to be removed or not would be expensive. On the other hand, erasing all indirect implications would be a waste of many implications which are still valid. We found a property of indirect implications in our algorithm which localizes the region where invalid implications may appear. Thus, only this region needs to be cleaned.

Let $l$ be a redundant line which is found in one of the runs for the vertex $v_{base}$. We define two regions, $R_1, R_2 \subseteq V$, as follows. $R_1$ consists of all vertices which can be reached from $l$ without passing through any vertex $w$ with no indirect implication, i.e. such that $indirect\_implication(w, 0) = \emptyset$ and $indirect\_implication(w, 1) = \emptyset$. $R_2$ is empty if $index(v_{base}) < index(u)$, where $u$ is the vertex fed by $l$. Otherwise, $R_2$ consists of all vertices whose indexes are in the interval between $index(u)$ and $index(v_{base})$.

*Theorem 2:* An indirect implication $(v = i) \rightarrow (u = j)$ can be invalid only if either $v \in R_1$, or $u \in R_2$, for some $i, j \in \{0, 1\}$.

It follows from the above theorem that, in order to make the implications consistent, it is sufficient to visit every vertex $v \in R_1$ and remove all implications stored with $v$, i.e. set $indirect\_implication(v, j) = \emptyset$ for all $j \in \{0, 1\}$. In addition, set $invalid\_implication(u) = on$ for all $u \in R_2$. The updating is done by the procedure UPDATEIMPLICATIONS.

Our experience is that indirect implications usually do not spread many levels forward from the level of currently processed vertex $v_{base}$. Therefore, $R_1$ is quite small. Normally, it has a constant size regardless of the circuit size. The region $R_2$ is almost always empty, because $index(u)$ is less than $index(v_{base})$ for 1 out of 1000 found redundancies on average. As a result, the updating process is very quick.

Note, that not all implications which we remove are invalid. However, checking their validity would take a considerable amount of time, since we do not keep track of how they were found. So, by removing them all, we save time. Furthermore, since $R_1$ and $R_2$ are small, most of valid indirect implications are left after updating.

*2) Identification of duplicated functions:* The presented algorithm is able to identity some vertices which implement equivalent or complemented functions, as well as constant vertices. Therefore, it can remove some redundancies which cannot be found by FIRE or ATPG.

Equivalent functions are identified as follows. If the assignment of the vertex $v_{base}$ to the value $i$ causes some gate $u$ to be set to the value $i$, as well as the assignment of $v_{base}$ to $\bar{i}$ causes $u$ to be set to $\bar{i}$, then we conclude that $v$ and $u$ implement the same function and add $u$ to the equivalence class of $v_{base}$, $E(v_{base})$.

Similarly, to identify complemented functions, $(v_{base} = i) \rightarrow (u = \bar{i})$ and $(v_{base} = \bar{i}) \rightarrow (u = i)$ imply that $v_{base}$ and $u$ are complements of each other. The vertex $u$ is added to the equivalence class of $\bar{v}_{base}$, $E(\bar{v}_{base})$.

Finally, all vertices in $E(v_{base})$ and $E(\bar{v}_{base})$ are replaced by a member of the class which is closest to the primary inputs in topological order (i.e. has the shortest longest path to the primary inputs). Let $u$ be the member selected from $E(v_{base})$ and $w$ be the member selected from of $E(\bar{v}_{base})$. If both $u$ and $w$ have more than one input, then the one which is further away from the primary inputs is substituted by an inverter fed by the other one.

To identify constant functions, we use the following simple property. If $(v_{base} = 0) \rightarrow (u = i)$ and $(v_{base} = 1) \rightarrow (u = i)$, then the vertex $u$ is the constant $i$.

Note, that redundant vertices are identified with a minimum search, by re-using the information available from the algorithm's flow.

## VI. EXPERIMENTAL RESULTS

This section compares the performance of the presented algorithm to the ATPG-based approach from [9]. We also show the results of FIRE [12] as a reference. Note that FIRE only identifies redundancy, but does not remove it.

Table I summarizes the results for ISCAS'85 benchmark set. Columns 2-4 show the results for FIRE, taken from [12]. Column 2, # red, is the total number of identified redundant lines. The sign '-" means that no result is reported in [12]. Column 3, % red, is the percentage of identified redundant lines compared to ATPG [9], and Column 4 is CPU time, in seconds. According to [12], FIRE was run on a SUN SPARC2. No parameters of the SUN SPARC2 machine are provided in [12], so a comparison of CPU times of the two algorithms is hard to make. Therefore, in order to evaluate the effect

| name | FIRE (results from [12]) | | | Presented without 4 improvements from Section V | | | Presented | | | ATPG [9] | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | # red | % red | $t$, sec | # red | % red | $t$, sec | # red | % red | $t$, sec | # red | $t$, sec |
| C17 | - | - | - | 0 | - | 0.01 | 0 | - | 0.00 | 0 | 0.00 |
| C432 | - | - | - | 45 | 60.8 | 0.01 | 63 | 85.1 | 0.00 | 74 | 4.43 |
| C499 | - | - | - | 0 | 0 | 0.01 | 0 | 0 | 0.01 | 8 | 0.27 |
| C880 | - | - | - | 0 | 0 | 0.01 | 0 | 0 | 0.01 | 8 | 0.08 |
| C1355 | - | - | - | 0 | - | 0.02 | 0 | - | 0.01 | 0 | 0.82 |
| C1908 | 6 | 15.4 | 1.8 | 24 | 61.5 | 0.02 | 26 | 66.7 | 0.02 | 39 | 0.51 |
| C2670 | 29 | 16.3 | 1.5 | 56 | 31.5 | 0.03 | 64 | 36.0 | 0.03 | 178 | 0.53 |
| C3540 | 93 | 38 | 11.9 | 144 | 58.8 | 0.08 | 167 | 67.9 | 0.09 | 246 | 2.17 |
| C5315 | 20 | 14 | 2.8 | 36 | 25.2 | 0.05 | 62 | 43.4 | 0.05 | 143 | 1.45 |
| C6288 | 33 | 82.5 | 1.3 | 69 | 98.6 | 0.06 | 113 | 161.4 | 0.06 | 70 | 2.00 |
| C7552 | 30 | 7.39 | 4.7 | 99 | 24.3 | 0.10 | 124 | 30.5 | 0.11 | 406 | 5.25 |
| **average** | 35.2 | 28.9 | 4.00 | 43 | 40.1 | 0.036 | 56.3 | 54.6 | 0.037 | 105.5 | 1.59 |

TABLE I: Benchmark results for ISCAS'85 circuits; average is computed for non-"−" entries.

of four improvements described in Section V, we re-run the presented algorithm with these improvements switched off. The presented algorithm without these improvements can be considered as our re-implementation of FIRE. The results are shown in Columns 5-7. As we can see, runtime improvements fully compensate the time consumed by quality improvements. If runtime improvements are switched off, while quality improvements are on, the presented becomes about 20% slower.

Columns 8-10 and 11-12 show the corresponding numbers for the presented algorithm and ATPG [9]. The experiments were run on a PC with Pentium III 750 MHz processor and 256 Mb memory. On average, for ISCAS'85 benchmarks, the presented algorithm removes 54.6% of ATPG redundancies using only 2.28% of its runtime.

On a larger set of 183 circuits from IWLS'02 benchmarks set with the average size of 780 gates and the maximum size of 25000 gates The presented algorithm without four improvements removes 14% of ATPG's redundancies, while with improvements it removes 19.3% of ATPG's redundancies, both using 2.5% of ATPG's time. These results do not include two benchmarks from the set, *spla* and *pdc*, for which ATPG did not finish in 2 hours, while the presented algorithm finished in 15 sec. As we can see, the proposed improvements allow us to find 37% more redundancies without increasing the runtime.

## VII. Conclusion

This paper presents a heuristic algorithm which efficiently identifies and removes redundancy in combinational circuits. Unlike other extensions of FIRE, the presented algorithm provides better quality of results without trading it for runtime. The speed of our heuristic makes it suitable for running multiple times during synthesis.

A possible extension of the presented approach is to employ an alternative strategy for removing redundant lines. In our current implementation, we used first-found first-removed approach in order to keep the runtime to minimum.

## References


[1] R. K. Brayton and C. McMullen, "The decomposition and factorization of Boolean expression," in *Proceedings of the IEEE International Symposium of Circuits and Systems*. IEEE, 1982, pp. 49–54.

[2] J. Darringer, W. Joyner, L. Berman, and L. Trevillyan, "Logic synthesis through local transformations," *IBM Journal on Research and Development*, vol. 25, no. 4, pp. 272–280, July 1981.

[3] E. Sentovich and D. Brand, *Flexibillity in logic*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[4] H.-C. Liang, C. L. Lee, and J. E. Chen, "Identifying untestable faults in sequential circuits," *IEEE Des. Test*, vol. 12, no. 3, pp. 14–23, 1995.

[5] M. Abramovici and M. A. Iyer, "One-pass redundancy identification and removal," in *Proceedings of the IEEE International Test Conference on Discover the New World of Test and Design*. Washington, DC, USA: IEEE Computer Society, 1992, pp. 807–815.

[6] A. D. Friedman, "Fault detection in redundant circuits," *IEEE Transactions on Electronic Computers*, vol. EC-16, pp. 99–100, February 1967.

[7] M. H. Schulz and E. Auth, "Improved deterministic test pattern generationwith application to redundancy identification," *IEEE Transactions on Computer-Aided Design*, vol. 8, pp. 811–815, July 1989.

[8] M. Berkelaar and K. M. van Eijk, "Efficient and effective redundancy removal for million-gate circuits," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. IEEE Computer Society Press, 2002, p. 1088.

[9] D. Brand and R. Damiano, "In the driver's seat of BooleDozer," in *Proceedings of International Conference on Computer Design*. IEEE, October 1994, pp. 518–521.

[10] M. Harihara and P. R. Menon, "Identification of undetectable faults in combinational circuits," in *Proceedings of International Conference on Computer Design*. IEEE, October 1989, pp. 290–293.

[11] P. R. Menon and H. Ahuja, "Redundancy removal and simplification of combinational circuits," in *Proceedings of VLSI Test Symposium*. IEEE, April 1992, pp. 268–273.

[12] M. A. Iyer and M. Abramovici, "FIRE: A fault-independent combinational redundancy identification algorithm," *IEEE Transactions on VLSI Systems*, vol. EC-16, pp. 295–301, June 1996.

[13] K.Gulrajani and M. S. Hsiao, "Multi-node static logic implications for redundancy identification," in *Proceedings of Design, Automation and Test in Europe Conference*. IEEE, March 2000, pp. 729–735.

[14] M. S. Hsiao, "Maximizing impossibilities for untestable fault identification," in *Proceedings of Design, Automation and Test in Europe Conference*. IEEE, March 2002, p. 949.

[15] V. C. Vimjam, M. Syal, and M. S. Hsiao, "Testing: Untestable fault identification through enhanced necessary value assignments," in *Proceedings of the 15th ACM Great Lakes symposium on VLSI*. IEEE, 2005.

[16] J. P. Kim, M. Silva, H. Savoj, and K. A. Sakallah, "RID-GRASP: Redundancy identification and removal using GRASP," in *Proceedings of International Workshop on Logic Synthesis*. IEEE, May 1987.

[17] A. Kuehlmann, M. Ganai, and V. Paruthi, "Robust Boolean reasoning for equivalence checking and functional property verification," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, December 2002.

[18] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proceedings of the 34th ACM/IEEE Design Automation Conference*, Anaheim, CA, June 1997, pp. 263–268.